STATIC EXCEPTION CHECKER FOR JAVA PROGRAMS

An Undergraduate Research
Option Thesis
Presented to
The Academic Faculty

By

Liubov Nikolenko

In Partial Fulfillment
of the Research Option
in College of Computing

Georgia Institute of Technology

December 2017

STATIC EXCEPTION CHECKER FOR JAVA PROGRAMS

Approved by:

William Harris
College of Computing
Georgia Institute of Technology

Dr. C. Karen Liu
College of Computing
Georgia Institute of Technology

ACKNOWLEDGEMENTS

I would like to thank David Heath, Caleb Voss, Collin Richards and Justin Nieto for their hard work on the project. I would also like to thank Prof. William Harris for his excellent metorship of this research study.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The purpose of this study is to present a tool, that can check if programs, compiled to Java Virtual Machine bytecode, are exception safe. The tool performs static analysis of input programs by reducing the programs to Horn Clause systems and solving the generated systems in the automatic theorem prover. The current version of the tool is publicly available in the Bitbucket repository. The tool can be used for custom constraint verification, safety checking and Horn Clause reduction.

CHAPTER 1
INTRODUCTION

Static program analysis is the analysis of computer software without executing its code. This technique is used for verifying certain properties of the given code and finding errors before runtime. The primary advantage of this approach is that it is does not require generating exhaustive test cases and running the program multiple times with different inputs, which can be non-trivial and time consuming. Today static analysis tools are widely used in the industry for tracking simple bugs and enforcing coding standards.

There are several implementations of static analysis of Java source code. Such tools as Find Bugs[1] and JLint[2] focus on the syntactic analysis of the program and match source code to known patterns of suspicious programming practice. Extended Static Checking System for Java[3] implements a model checking approach to verify a given program against user-defined loop invariants. Overall, the majority static analysis tools for Java are able to detect simple bugs (such as undeclared variables), enforce syntactic standards and check the code against simple invariants (e.g. some variable is greater than zero before and after a loop).

However, there is no software that can statically analyze a program and identify if it throws a runtime exception, which is not related to a simple syntax error, but rather caused by the pitfall of the algorithm. Software that analyses the code syntactically cannot find complex errors, while current tools that use model checking require users to define their own constraints, which can become very challenging and code-specific if they are trying to model runtime exceptions. In addition, checking for exceptions becomes non-trivial in the presence of all features of Java Virtual Machine programming languages such as branches, method calls and sub-classing.

This research project attempts to address the outlined issue in the static analysis field

by developing a tool that will use model checking to prove or disprove that a given Java Virtual Machine code snippet can produce runtime exceptions. In particular, this project is targeting Array Index Out of Bounds Exception, Null Pointer Exception, Class Cast Exception, and Division by Zero Exception. This research will model the listed exceptions using Horn clauses, translate the given code into the same form and use an automatic theorem prover to check if these exceptions are possible in the given code.

This paper provides an overview of two major approaches to static program analysis and a summary of currently existing tools that are able to analyze code statically in the Literature Review section. The implementation section guides the reader through the structure of our application and provides pointers to open source libraries that were used to build the project. The pointer to the current project code base and the summary of tool functionality is provided in the Results section, while the Discussion section lists multiple applications of the developed tool and points out the limitations of the project.

CHAPTER 2
LITERATURE REVIEW

Static program analysis approach is mostly used for verifying certain program properties of the code. Static analysis is very efficient, because it allows users to bypass manual testing of numerous inputs. Static analysis applications need to be executed only once in order to infer program properties.

Static program analyses are divided two major areas: syntactic analysis and semantic analysis. Syntactic analysis checks the text in the provided source code for syntax errors, while semantic analysis involves an in-depth processing of the program and derivation of its properties.

This project focuses on developing a tool that will semantically analyze programs before runtime, written in languages that can be compiled to Java Virtual Machine (JVM) bytecode such as Java, Scala and Python. In particular, the application will prove or disprove that a given JVM code can throw an exception: an error in the program that disrupts its normal flow and leads to an undefined outcome. This project is targeting Java exceptions that are caused by accessing an invalid memory address in the given context, performing incompatible data type conversion and diving by zero.

2.1 Syntactic Analysis

Currently, the majority of static program analysis tools for Java employ syntactic analysis and focus on enforcing coding standards or detecting syntax errors in the code. Such applications as Find Bugs [1], JLint [2] and PMD [4] verify that each line of a given code segment is syntactically correct and does not contain unsafe programming practices. This approach is useful for identifying simple bugs before runtime. However, it does not analyze the code in depth, and, as a result, can produce false warnings that actually do not

correspond to errors in the code. In addition, syntactic code analysis is not able to check the code for more complex errors such as accessing an invalid memory address. Overall, syntactic code analysis is effective only when it comes to detection of simple bugs in the code.

2.2 Model Checking

Model checking is a technique used for semantic program analysis. This approach addresses a problem of verifying certain properties in a given program. Lazy Abstraction with Interpolants [5] algorithm uses Craig Interpolants (logical formulas that are constructed from a limited set of variables) to model relationships between program variables. The outlined procedure generates a set of logical expressions at given program point and compares them to statements generated at previous program points in order to construct a Craig Interpolant that proves or disproves that a certain program point is reachable. The algorithm can apply these statements multiple times, if they show up in multiple paths. Also, it builds invariants by analyzing infeasible paths. This procedure is very efficient, because it generates statements only for one program point at a time and, therefore, does not require analyzing program paths that are not taken. Moreover, Lazy Abstraction does not require the strongest statements about each program point, which allows reusing the same statements for different program paths and reduces processing time. Even though this algorithm has been implemented in Z3 [6] as Duality Horn Clause Solver [7], the tool has not modeled Java semantics. Lazy Abstraction algorithm provides an answer to a model checking problem, which cannot be applied to a Java code snippet.

Extended Static Checking System for Java [3] also uses model checking for testing Java programs. It takes in preconditions, post conditions and loop invariants in the form of special comments from a programmer and uses an automatic theorem prover [6] to verify that the program matches provided specifications. This tool is useful for verifying loop invariants that involve variables in a given code (for example, verifying that a variable is

non-negative before and after the loop). It is highly customizable, because the end user is allowed to define any program constraints. However, coming up with constraints that model Java exceptions in a given environment is non-trivial and requires generating a different set of expressions for every program. Extended Static Checking System for Java verifies loop invariants, but it cannot be widely used by programmers for detecting exceptions, because it requires complex user input and, therefore, reduces efficiency benefits of static program analysis.

Bandera tool [8] partially tackled the problem of automatic exception checking without intensive user input. Bandera analyzes code for exceptions related to multithreading execution of multiple program paths at the same time. The tool uses model checking to verify that uncorrupted shared data will be accessible to all of the threads at some point of execution. However, Bandera cannot analyze Java library calls, which greatly limits the usability of this application. Overall, this tool targets issues that come with parallelism and omits analyzing sequential parts of the program in detail. In addition, Bandera cannot be used for debugging real-world applications, because the majority of the software contains some kind of Java library call.

2.3 Exception Modeling

Currently, there is no tool that can effectively verify that sequential Java Virtual Machine programs do not throw exceptions. Software that employs syntactic analysis cannot detect the majority of exceptions, while model checking tools require complicated user input that has to be designed for each program individually. This project will produce model checking software, which will automatically generate constraints for a given program that correspond to common Java exceptions and verify that those constraints hold in the provided code snippet. The resulting application will be able to determine if a program can throw an exception at some point of its execution.

CHAPTER 3

IMPLEMENTATION

3.1 High-Level Overview

The tool takes in any source code, that can be compiled using Java Virtual Machine (JVM) [9] as a parameter and transforms it into a bytecode. At this point the tool is concerned with analyzing bytecode. The assembly code will be translated into a set of Horn Clauses - logical expressions, that define relationships between program variables in the rule-like form (e.g $X \implies Y$). In addition, the tool generates assertions that involve variables from the input program to represent certain Java exceptions. The tool will test if two sets of generated expressions are mutually exclusive in The Z3 Theorem Prover [6]. Mutually exclusive sets of expressions correspond to an exception-safe code, while sets with non-conflicting expressions will indicate a possibility of an exception in the code.

Figure 3.1 provides an overview of safety verification process, implemented in the tool.

The outlined algorithm preserves and analyzes all properties of the program, due to generation of a separate Horn Clause at each program point. Therefore, this procedure will give valid conclusions about program safety.

The project is written in OCaml [10] programming language, due to its efficiency and availability of libraries for static analysis.

3.2 Representating Java ByteCode

Java bytecode, produced by Java compiler, is very tedious to manipulate, because it includes many JVM-specific details, such as stack-based implementation. In order to avoid dealing with low-level assembly details, the project took advantage of Sawja [11] - a module that translates JVM bytecode into OCaml types and data structures. Particularly, the

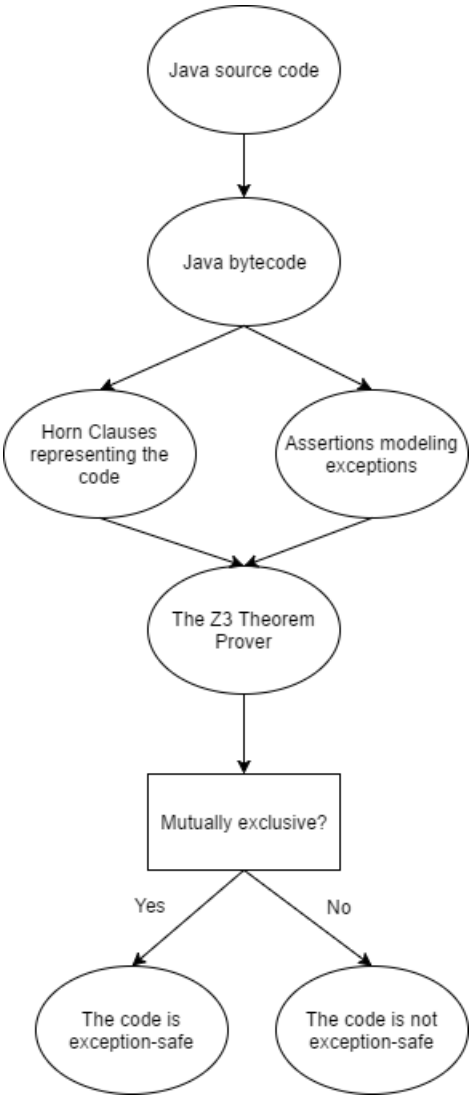

Figure 3.1: An overview of safety verification process.

tool uses methods from such modules as Javalib, JBasics, Sawja_pack and JProgram to obtain a Java bytecode representation, suitable for program analysis. In addition to parsing program statements, we also keep track of data types, classes and method calls.

3.3 Project Architecture

Once Sawja bytecode representation is obtained, it undergoes two more transformations. At first, the code is translated into intermediate language, defined in OCaml, that represents Horn Clauses. Then this intermediate representation is used for creating Z3 statements. As

a result, the project consists of three major layers, shown in Figure 3.2.

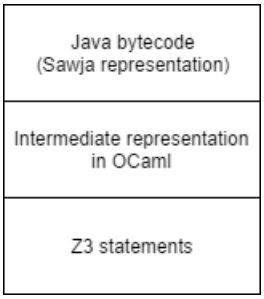

Figure 3.2: Three major layers of the project.

We chose to insert an additional layer between Sawja bytecode and Z3 call intentionally due to the following reasons:

1. It is possible to optimize an intermediate representation of the code before calling Z3.

2. It reduces the degree of dependence on Z3 and makes it easier to invoke another theorem prover if needed.

3. It provides an additional layer of abstraction, which isolates the programmer from Z3-specific details.

3.4 Intermediate Representation

In order to generate an intermediate representation of the code, the tool iterates over each program statement and determines Horn Clauses that represent a transition between program states, expressed in program variables.

Since Java bytecode has a finite number of instructions, we came up with a mapping of how each instruction alters previously generated predicate (a logical expression that can be either true or false). In addition, we are computing the next program state for each Horn Clause, derived from a given instruction. The resulting Horn Clause has the following

format:

$$Predicate_1 \wedge Predicate_2 \wedge ... \wedge Predicate_n \implies Next\ program\ state$$

Particular Java features such as if statements and loops require more than one Horn Clause in order to model all possible transitions. If statements produce a separate Horn Clause for each possible path of program execution. All boolean conditions from the if statement necessary to take a certain branch get added to the set of predicates, while the path in question gets represented as next program state in the Horn Clause. As a result, `if` statements get modeled as follows:

$$if(Condition_1)\ Path_1 \mapsto Condition_1 \implies Path_1$$

$$if(Condition_2)\ Path_2 \mapsto Condition_2 \implies Path_2$$

$$\vdots$$

$$if(Condition_n)\ Path_n \mapsto Condition_n \implies Path_n$$

Representing loops with Horn Clauses requires coming up with loop invariants - predicates that do not change throughout the course of the loop execution. The tool also establishes the termination condition of the loop - a condition that determines if and when the loop terminates. Combining these pieces of information together produces Horn Clauses in the format specified below:

$$(Predicates\ before\ the\ loop) \wedge \neg(Termination\ condition) \implies Beginning\ of\ the\ loop$$

$$(Loop\ invariants) \wedge \neg(Termination\ condition) \implies Beginning\ of\ the\ loop$$

$$(Loop\ invariants) \wedge (Termination\ condition) \implies End\ of\ the\ loop$$

An example in Figure 3.3 illustrates how a basic code snippet is represented as a set of Horn Clauses:

```
1  int i = 0;
2  while (i < N)
3      i += 2;
4  done;
5  assert (i % 2) == 0;
```

Figure 3.3: Initial pseudocode.

This pseudocode will result in the following Horn Clause representation:

$$i = 0 \implies R(i)$$
$$R(i) \wedge (i < N) \implies R(i+2)$$
$$R(i) \wedge (i \geq N) \implies i \mod 2 = 0$$

Figure 3.4: A set of generated Horn Clauses

3.5 Modeling Exceptions

Exceptions are modeled as a separate set of assertions over given program variables that needs to be verified. However, we ignore the properties, inferred from the program and focus on the details of the exception. We construct an expression, representing the exception based on two factors: the semantics of the exception itself and variables present in the program.

3.5.1 Division by Zero Exception

Every time there is a division in the code, the tool will generate an assertion that the divisor in the expression is equal to 0.

3.5.2 Array Index Out of Bounds Exception

Each access of an array (e.g. $element = A[i]$) in the code produces an assertion that $i >= N$, where N is the declared size of the array. The array size is maintained per each

instantiated array.

3.5.3 Null Pointer Exception

Whenever there is an operation or a method call on an object for the first time after declaration of modification, there is an assertion that $Object == NULL$

3.5.4 Class Cast Exception

Each time there is a cast of one class into another, there is an assertion that two classes are incompatible. Consider the following code fragment:

```
1 Class1 a = new Class1();
2 Class2 b = (Class2) a; //class casting takes place
```

The generated assertion corresponds to this pseudocode:

$$assert(Class2\ (is\ not\ a\ parent\ of)\ Class1)$$

Class hierarchy will be derived from Sawja bytecode representation.

3.6 Representation for a Theorem Prover

We are using The Z3 Theorem Prover as an engine for solving Horn Clauses. Therefore, the intermediate representation of the code is converted into Z3 programming language. In particular, the tool takes advantage of such features as assertions, propositional logic and iterations over a set of possible solutions. The tool has to be run in `set-logic HORN` mode. Each operation present in the intermediate representation of the code is mapped to a number of statements in Z3. Transitions between program states are translated into invariants, defined as function declarations (predicates in the Horn Clause), boolean expressions, and logical implications (next program state in the Horn Clause) between them. Exceptions are modeled as assertions that involve program variables.

As an example, Horn Clauses from Figure 3.4 get translated into the following Z3 statements:

```
1  (set-logic HORN)
2  (declare-fun R (Int) Bool)
3
4  (assert
5    (R 0))
6
7  (assert
8    (forall
9      ((i Int) (N Int))
10     (=>
11       (and
12         (R i)
13         (< i N))
14       (R
15         (+ i 2)))))
16
17 (assert
18   (forall
19     ((i Int) (N Int))
20     (=>
21       (and
22         (R i)
23         (not
24           (< i N)))
25       (=
26         (mod i 2)
27         0))))
28
29 (check-sat)
30 (get-model)
```

Since there is a Z3 module for Ocaml [12], we could easily integrate Z3 call into the tool. Z3 takes in generated Horn Clauses and assertions and automatically solves a given system of logical constraints. If there a solution under which all of the Clauses are valid entailments, then the program is not exception-safe. However, if there is no solution to a given system of constraints, then the code will not throw exceptions in question at any point of its execution.

CHAPTER 4

RESULTS

The current implementation of the project is publicly available at the Bitbucket repository [13].

The resulting tool has the following functionality:

- Translating the code, that can be compiled to the JVM bytecode, into a system of Horn Clauses, defined in terms of variables, present in the program.
- Checking the input code against custom assertions, specified by the user with the `Assert` statement.
- Checking if the code can produce Array Index Out of Bounds Exception, Null Pointer Exception, Class Cast Exception and Division by Zero Exception.

After analyzing the input code, the tool outputs `sat` or `unsat` result. `sat` statement means that the program may produce an exception at some point of its execution, while `unsat` implies that the program is exception safe.

In order to observe the functionality of the tool, consider checking the code snippet in Figure 4.1 for Division by Zero Exception. The tool produces `unsat` output, because there is no integer y such that $5 * y + 1 = 0$, while the expression $5 * y + 1$ is the only source of potential exception. On the other hand, similar analysis of code snippet on Figure 4.2 results in `sat`output in the tool, indicating that the program is not exception-safe. Indeed, if $x = 2$, then $y = 1$ and $5 * y - 5 = 5 * 1 - 5 = 0$.

The set of test cases that were used to evaluate the correctness of the tool is available in the test folder in the repository. In particular, the tests contained linear programs, conditional programs, function calls and loops.

```
1  int y;
2  int z;
3  //x is a variable, that was defined in the code earlier
4  if (x % 2  == 1){
5      y = x + 1;
6  }
7  else{
8      y = x / 2;
9  }
10 z = 10 / (5 * y + 1);
```

Figure 4.1: Initial Java code

```
1  int y;
2  int z;
3  //x is a variable, that was defined in the code earlier
4  if (x % 2  == 1){
5      y = x + 1;
6  }
7  else{
8      y = x / 2;
9  }
10 z = 10 / (5 * y - 5);
```

Figure 4.2: Modified Java code

Since the tool has successfully passed all of the provided test cases, it can be inferred that the project is able to parse and analyze the Java semantics, outlined above.

CHAPTER 5
DISCUSSION

5.1 Safety Verification

The resulting application provides static checking against Array Index Out of Bounds Exception, Null Pointer Exception, Class Cast Exception, Division by Zero Exception, and user-defined constraints. The tool automatically generates all the necessary constraints that model the input program as well as safety statements, allowing the programmer to bypass manual analysis and constraint generation for each individual program. The tool provides a new level of automation for code testing, since the user does not have to provide any input to the tool, besides the program that needs to be tested, while previous static analysis tools did not support exception-modeling semantics [1, 2, 4] or required additional user input [3]. As a result, the tool can significantly increase the speed of code testing during software development.

5.2 Horn Clause Generation

The tool reduces the input program to Horn Clause system before checking if the program is exception-safe. The functionality of reducing a program to a set of Horn Clauses can be used separately from the rest of the tool for other applications of model checking, such as analysis of information flow in the program and proof of equivalence of two programs.

Information flow analysis may employ Hoare-style framework to represent total or partial transfer of information from highly secure variables to the program output [14]. Barthe et al. represents secure information flow by self-composition - a procedure that constructs a program, which is identical to the input program, but with different variable names, models both programs as one system of Horn Clauses and makes an assertion that equivalent

inputs produce equivalent outputs [15]. Other researchers that work on information flow analysis similarly use approaches that involve constructing a Horn Clause system from a given program [16, 17].

Proving that two input are equivalent (if given two equivalent inputs, they are guaranteed to produce equivalent outputs) also involves solving Horn Clause systems.De Angelis et al. developed algorithms that verify relational properties between two programs by transforming Horn Clause representations of the programs [18, 19].

Therefore, modeling an input program as a set of Horn Clauses is a common technique used in various applications of static analysis. Since the developed tool can be used as a standalone Horn-Clause generator, the applications of the project go beyond safety checking. In particular, our tool can produce Horn Clauses that can be used for other kinds of static analysis.

5.3 Limitations

The current version of the tool has several limitations. The tool was not extensively tested and it is not known if it can be used on the real-life programs. In addition, the current automatic theorem prover used for the tool [6] is not able to solve non-linear Horn Clause systems. For instance, if the input program is reduced to a non-linear system, the tool may not produce a concrete `sat` or `unsat` and return `unknown` instead, meaning, that it is not able to verify if the given program is exception-safe. The project also does not support the analysis of multithreaded programs, since accesses of the shared data and lock semantics are not modeled by this tool.

5.4 Future Work

The project was tested only on trivial toy programs, that are shorter than 100 lines of code. Therefore, the tool requires more extensive testing that employs complex real-life benchmarks. Specifically, the project needs more testing on programs that involve non-linear

arithmetic in order to determine the possible limitations of the project. In addition, the performance of the tool needs to be evaluated using large benchmarks that contain several thousands lines of code, in order to analyze if the tool's performance on the enterprise-level programs.

REFERENCES

[1] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, "Improving your software using static analysis to find bugs," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ACM, pp. 673–674, ISBN: 159593491X.

[2] JLint. `http://artho.com/jlint`.

[3] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Pldi 2002: Extended static checking for java," *ACM Sigplan Notices*, vol. 48, no. 4S, pp. 22–33, 2013.

[4] PMD/Java. `http://pmd.sourceforge.net`.

[5] K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, T. Ball and R. B. Jones, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 123–136, ISBN: 978-3-540-37411-4.

[6] L. De Moura and N. Bjrner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 337–340.

[7] K. L. McMillan and A. Rybalchenko, "Solving constrained horn clauses using interpolation," *Tech. Rep. MSR-TR-2013-6*, 2013.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, and H. Zheng, "Bandera: Extracting finite-state models from java source code," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, IEEE, pp. 439–448, ISBN: 1581132069.

[9] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education, 2014, ISBN: 013390590X.

[10] Ocaml. `http://www.ocaml.org`.

[11] L. Hubert, N. Barr, F. Besson, D. Demange, T. Jensen, V. Monfort, D. Pichardie, and T. Turpin, "Sawja: Static analysis workshop for java," in *Formal Verification of Object-Oriented Software: International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, B. Beckert and C. March, Eds. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2011, pp. 92–106, ISBN: 978-3-642-18070-5.

[12] Module Z3. http://z3prover.github.io/api/html/ml/Z3.html.

[13] Bitbucket repository, containing the tool. https://bitbucket.org/wrharris/safety-itps/src/11811f33178ee4b403ec0f4713073a45e3ec3b33/?at=horn-interpolation.

[14] Q.-S. Phan, "Self-composition by symbolic execution," 2013.

[15] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, IEEE, pp. 100–114, ISBN: 076952169X.

[16] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *International Static Analysis Symposium*, Springer, pp. 352–367, ISBN: 3540285849.

[17] B. Blanchet, "Using horn clauses for analyzing security protocols," *Formal Models and Techniques for Analyzing Security Protocols*, vol. 5, pp. 86–111, 2011.

[18] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, "Verifying relational program properties by transforming constrained horn clauses,"

[19] ——, "Relational verification through horn clause transformation," in *International Static Analysis Symposium*, Springer, pp. 147–169.